Autonomy and Robotics Area
Computational Sciences Division
M/S 269-3
NASA Ames Research Center
Moffett Field, CA 94301
POC: kurien@ptolemy.arc.nasa.gov

Livingstone Real Time Interface

Autonomy and Robotics Area
NASA Ames Research Center

http://ace.arc.nasa.gov/postdoc/livingstone

Table of Contents

Remote Agent Autonomy Toolbox Specification

Livingstone Realtime Interface

November 17, 1998

Introduction

This is a specification document, in progress, for Livingstone's Real-time Interface. This document includes specifications for:

- The real-time external interface,
- Regression testing and empirical evaluation framework,
- real-time interface internal data structures,
- Internal interfaces between major subsystems,
- Description of key algorithms and theorems,
- Performance statistics on representative examples.

Problem Statement

In the previous section we saw how to set up commands and monitors which act as an interface to the Livingstone model. We also introduced the functions do-cmd and do-monitors which would inject a command or set of monitors into the model and do a diagnosis. These functions are convenient for debugging a model but poor for actually running a physical system. They are synchronous, in that they take a command and all of the monitors that came with it, perform a diagnosis, then returns an answer. When controlling a real system, it may take several seconds or more for a command to take effect and all of the monitor changes associated with it to become available. In addition, while one command is being processed unrelated monitors which indicate an unrelated failures may become available, and we would like to diagnose that failure immediately.

To accomodate the asynchronous nature of commands an monitors being exchanged with a physical device Livingstone has a set of interfaces which allow the user to inject new commands and monitors into the system as they come in. Livingstone then tracks things such as whether an incoming monitor value is associated with a command which is currently executing or whether it represents a failure which must be diagnosed and reacted to immediately.

It provides a queue onto which you can place incoming commands and monitor values as well as requests for recovery. These items are processed by Livingstone and any recovery actions or changes in system state are sent back via two reporting functions that the user must define.

These functions can be integrated with Lisp code which communicates with the outside world via sockets, CLASH, CORBA and so on to allow Livingstone to interface with separate programs running on the same or another computer. CLASH, sockets and a variant of CORBA have all been tested and sample code is available.

Livingstone provides an interface for easing the connection of Livingstone to an external real-time process (an external program, another Lisp thread, etc).

This interface provides a message queue which the external process uses to inform. Livingstone of incoming commands and monitor values, and sends requests to Livingstone for state information and recover plans. These items are processed by Livingstone and any recovery actions or changes in system state are sent back via two reporting functions that the user must define.

Interaction Cycle Between Livingstone and Real-time

The basic interaction cycle consists of:

1. `spawn-queue-dispatcher.` This creates a queue to send information and requests to Livingstone.

2. As monitor data becomes available the real-time system, calls `queue-monitor-value` to specify each monitor value, followed by `queue-start-monitors-and-time,` to tell Livingstone to process these monitor values, and by what time these values will stabilize.

3. As commands are given, call `queue-start-command-and-time.` This informs Livingstone that a command has been invoked, and the time by which the command's effects will have stabilized.

4. As Livingstone performs mode identification, it will report any state changes inferred from the monitor data and commands by invoking `report-transitions,` a function provided by the real-time user.

If a recovery is needed, the real-time system calls queue`-generate-recovery` with a specification of the goals to be achieved and constraints to be maintained. Livingstone returns the recovery plan by invoking `report-recovery-actions,` a function provided by user.

Interface

Startup

`spawn-dispatcher` *[Function]*

This function creates the queue for mediating messages between the real-time and Livingstone. It must be called before any of the queueing functions below are called.

This function starts a second thread that will run the actual Livingstone inference engine. The calling thread returns immediately, and can be used to call the QUEUE- functions described below to inject commands and observations into the model.

`queue-request-full-state` *`id`* *[Function]*

Reports all Livingstone state information via `report-transitions`. This function can be called at any time after `spawn-dispatcher` but it is typically called before any commands are given to ensure Livingstone and other software components are synchronized as to the current state of the controlled system.

Commanding

`queue-start-command-and-time command value id &key` *`time system`* *[Function]*

Called when a new command is about to be issued by the real-time. Livingstone starts a timeout for command completion to occur after *`time`* delay, where *`time`* defaults to `*default-command-timeout*`. It then accumulates any changes in monitor values during that time and after the delay determines the next state (diagnosis). It reports any relevant state changes via `report-transitions`.

The purpose of the time delay is to make sure that the control system stabilizes, before trying to determine the next state – this avoids race conditions. also processes monitor changes during the time interval, to Any monitor changes that come in before the specified time expires which Livingstone predicts should change during the execution of the command will not cause a diagnosis. If the monitor are not expected to change as a result of the command, then Livingstone will start a separate diagnosis and return the results via `report-transitions.` and *`system`* defaults to `*system*`.

Asserting Monitoring Information

`queue-monitor-value` *`monitor value`* *[Function]*

Call to stack up monitor values for processing by the function below (e.g., if a data packet with 4 monitor values are received).

`queue-start-monitors-and-time` *`id`* `&key time system` *[Function]*

Call when a set of monitor values is ready for processing. That is, if you receive data for 4 monitors from some external source, call `queue-monitor-value` for each, then call `queue-start-monitors-and-time`. Livingstone determines if the monitor changes are expected because of a command which is currently running and does not start separate processing for those. If the monitor changes were not expected, Livingstone starts a timeout of the specified time, does a diagnosis, and reports via report-transitions. *`time`* defaults to `*default-command-timeout*`, and *`system`* defaults to `*system*`.

Recovery

`queue-generate-recovery` *`recovery-request id`* *[Function]*

Generates a recovery which when executed will put the modeled system into a state which entails the recovery request. The recovery is reported via `report-recovery-actions`.

id will be passed back when the recovery actions are reported and can be any integer.

recovery-request is a list of proposition-monitor values which must be achieved by the recovery. The recovery request is in terms of proposition-monitors instead of raw Livingstone propositions so that small changes to the implmentation of the Livingstone model will not break the interface.

Note the tuples valid for `report-transitions` and `queue-generate-recovery` are declared inside of Livingstone as `proposition-monitors`.

Required Response Functions

This interface assumes that the following functions are defined:

`report-transitions` *`transitions prop-transitions unique-id`* *[Function]*

This function takse the list of transitions resulting from an event, the prop monitor transitions, and the unique integer `ID` of an event. It should be defined by the user such that it sends the transitions off to whatever piece of software is interested in them.

unique-id will be the same id as passed with the command or monitor insertion that caused the state change.

transitions will be a list of transitions. Each transition is a list of the previous mode of a component and the current mode of a component. The mode is a structure specific to an instance of a component and has a pointer back to the specific component of which it is a part.

prop-transitions will be a list of the prop monitor values which became true as a result of the transition.

`report-recovery-actions` *`recovery-exists recovery-action unique-id`* *[Function]*

recovery-exists will be T if a recovery could be found.

unique-id will be the same id as passed with the recovery request.

recovery-actions will be a list of recovery actions. Each recovery will be a list of a transition name from the model (eg turn-off) and the specific component mode that should undergo the transition. The mode is a structure specific to an instance of a component and has a pointer back to the specific component of which it is a part. If the recovery requested was entailed by the current state of the system *recovery-actions* will be NIL.

Parameters

`*default-command-timeout*` *[Parameter]*

Default timeout when waiting for quiescence after a command is executed. The default is 5.0 seconds.

`*default-monitor-timeout*` *[Parameter]*

Default timeout when waiting for quiescence after a monitor is received. The default is 3.0 seconds.

Other Functions

Utilities

`unkeywordize` *`item`* `&key` *`package`*

Takes an atom or list and converts any keywords to symbols in the package. This is useful because the model usually is not constructed with keywords, and any symbols in it are in the `tp` package. Symbols that come out of CLASH or other communication packages are often keywords. *`package`* defaults to `:tp`.

Cancelling commands

The following functions are only needed in special cases where we know that a command may not get invoked. In this case we do not want to start a command and automatically set up a timeout for its termination. An example is a command that passes over a bus and might not actually get to the device.

`Queue-start-command` will do the processing to set up the command. If the command reaches the destination, a timeout can be set up to end the command. If not, `abort-command` will undo the processing performed by `queue-start-command` to set up the command.

```
queue-start-command command value

queue-abort-command command value
```

Example Session

Monitor data available from the hardware will most likely be in the form of some real valued or integer measurements from the system. Since Livingstone monitors operate variables with a small number of discrete values there will need to be some filtering code which converts the actual measurements into ranges. This code can be simple (eg thresholding into positive, negative and zero) or quite complex. This filtering could be done inside of the Lisp image in which Livingstone runs, meaning the raw values are transported into Lisp, or or outside of it, in which case discrete values are sent.

Case 1:

The low level control software of the hardware reads a number of sensors on the hardware, packages up the data into one packet and sends it to Lisp. In Lisp, a function determines which part of the packet corresponds to the measured position of switch one, converts that value from 0/1 to off/on. The function determines if the value is different from the last received value and if so calls

(queue-monitor-value '(switch-sensor switch1) 'on)

After all relevant information is extracted from the data packet, the following call is made to begin processing of the monitors (*id* and *timeout* values are representative only).

(queue-start-monitors-and-time 999 :time 3)

Case 2:

The low level control software of the hardware reads a number of sensors on the hardware, packages up the data into one packet and sends it to Labview. Labview does some filtering on the data to eliminate transients or detect some interesting feature. It then converts the real data into the discrete values Livingstone expects. Labview then sends an encoding of the discrete value and the Livingstone monitor it is associated with through a socket to Lisp (eg "(stable (current-draw switch1))")

Inside Lisp the data from the socket is decoded and used to call

(queue-monitor-value '(current-draw switch1) 'stable)

(queue-start-monitors-and-time 999 :time 3)

Regression Testing and Empirical Evaluation Support

To Do:

- To be written.

Real-time Interface Internal Objects

***

Submodule Interfaces

The real-time interface is decomposed into *** subsystems:

- *** ***

The following subsections define the *current* interfaces of each subsystem, respectively. At this point the division is still fluid, and does not constitute a design specification. However, a submodule design specification is important in terms of sharing code between different implementations.

Some of the submodule interface functions are part of the real-time external interface, or part of the object specifications. Detailed comments are included here for only those functions that are not already part of the external real-time interface and object specifications.

Debugging

Key Algorithms and Formal Analysis

Performance Analysis